Funciones y ciclos (Parte I)

Funciones

Competencias

- Distinguir el concepto de las funciones y sus estructuras para aplicar correctamente sus características.
- Codificar una rutina JavaScript aplicando las diferentes formas de declarar funciones para resolver el problema planteado.

Introducción

Una de las tareas más comunes en programación es automatizar procesos, esto significa que tareas que son repetitivas se pueden realizar "solas" con ayuda de un computador y un par de líneas de código. Bajo esta misma premisa es que nacen las funciones, las cuales nos permiten agrupar un bloque de código bajo un nombre que contiene entradas y ofrece un valor de retorno.

La ventaja de utilizar funciones en nuestros desarrollos, es encapsular porciones de código y llamarlas en cualquier momento desde nuestro programa; de esta manera, evitamos escribir código una y otra vez, agilizando el desarrollo y facilitando su legibilidad y orden.

Funciones

Existen dos tipos de funciones, las que vienen incluídas en JavaScript (también llamadas *built-in functions*), como por ejemplo console y Math:

```
console.debug(Object);
console.info(Object);
console.error(Object);
console.log(Object);

Math.random();
```

Y las otras, son las que podemos definir nosotros:

```
function hola(nombre){
     console.log(`Hola ${nombre}.`)
}

hola('Javier');    // => Hola Javier
```

Según la página de desarrolladores de Mozilla Firefox, una función es un procedimiento en JavaScript, es decir, un conjunto de sentencias que realizan una tarea o calculan un valor, o sea, una función será una porción de código con un fin definido y establecido por el programador. Este objetivo puede ser reutilizable: procesar datos, recibir valores o regresar resultados, convirtiendo el código en pequeños módulos con funciones definidas.

Estructura de una Función

Una función se compone de cuatro elementos que si bien no son todos obligatorios, si son primordiales para su funcionamiento: **un nombre, argumentos, parámetros, un bloque de operaciones y un retorno**, estos se ordenan de la siguiente manera:

```
function nombre (parametro1, parametro2, ...) {
    // operaciones
    return resultado;
}

nombre(argumento1,argumento2);
```

Para recordar y ejercitar los elementos estructurales de una función, puedes consultar el documento **Material Apoyo Lectura - Estructura de una función,** ubicado en "Material Complementario". En este documento podrás revisar en detalle aspectos teóricos y prácticos de este contenido.

Argumentos y Parámetros de una Función

En JavaScript, no es necesario definir el tipo de dato del parámetro como se hace en otros lenguajes de programación, ya que no revisa ni el número de parámetros (cantidad) ni los tipos de datos de los parámetros recibidos en una función. Esto significa que puedes pasar cualquier variable como argumento y recibirlos como parámetros sin importar el tipo de dato.

Retornos de una Función

Es posible guardar el valor retornado por una función a una variable utilizando el operador de asignación. Debido a que JavaScript no demanda conocer el tipo de dato de retorno, es posible retornar cualquier tipo de variable. Por ejemplo, en el siguiente código, se asigna el valor de retorno de la función `nombre` a `miVariable`:

```
function nombre (parametro1, parametro2, ...) {
    // operaciones
    return resultado;
}
var miVariable = nombre();
```

También se puede asignar cualquier tipo de retorno, esto incluye funciones. Por ejemplo, si queremos construir una función que depende de otra función:

```
const f = function saludar(nombre, nombreFuncion) {
    function hola() {
      return "Hola " + nombre;
    }

    function buenosDias() {
      return "Buenos días " + nombre + "!";
    }

    if (nombreFuncion === "hola") {
      return hola;
```

```
    }

    return buenosDias;
}

let miFuncionDeHola = f("Desafio", "hola");

// Ahora miFuncionDeHola es una función y se debe llamar como tal.
miFuncionDeHola();  // "Hola Desafío"
```

Por lo demás, JavaScript no exige que exista algún tipo de retorno en las funciones, una función sin el uso de **return** es totalmente válida:

```
function imprimirArgumento (parámetro) {
    console.log("El argumento es: " + argumento);
}
```

Ejercicio guiado: Suma de tres números

Desarrollar un programa en JavaScript que calcule la suma de tres números enteros e indique el resultado directamente en la función, es decir, sin retorno de valor alguno. Sigamos los siguientes pasos:

- ***Paso 1:*** Crear un archivo index.html y un archivo funciones.js dentro de una carpeta en nuestro sitio de trabajo.

- ***Paso 2*:** Al archivo index.html debemos agregarle toda la configuración básica de un documento HTML.

- ***Paso 3*:** En el archivo funciones.js debemos agregar el código necesario para inicializar las tres variables con números diferentes, luego crear la función llamada “sumaNumeros” que recibe como parámetros esos tres números, realice la suma y los guarde en una variable, para luego mostrar el resultado dentro de la misma función. Finalmente hacer el llamado a la función pasando como argumentos los tres números iniciados en las variables.

```
var num1 = 1;
var num2 = 2;
var num3 = 3;

function sumaNumeros (num1,num2,num3) {
    let suma = num1 + num2 + num3;
    alert("El resultado es: " + suma);
};

sumaNumeros(num1,num2,num3);
```

Siendo el resultado mostrado en la ventana emergente en el navegador:

Imagen 1. Resultado de la suma de tres números dentro de una función.
Fuente: Desafío Latam

Ejercicio propuesto (1)

Desarrollar un programa en JavaScript que calcule la resta de tres números enteros indicados en variables separadas previamente e indique el resultado directamente en la función de la resta, es decir, sin retorno de valor alguno.

Ejercicio propuesto (2)

Realizar un programa en JavaScript que calcule el volumen de un cilindro mediante una función. A dicha función se le deberán pasar los valores del radio y la altura. Luego, debe mostrar el resultado del volumen dentro de la misma función sin retornar ningún tipo de valor. (Ayuda: volumen de un cilindro es: PI*radio^2*altura).

Otras formas de desarrollar funciones

Es necesario mencionar que no todas las funciones poseen un nombre, sino que también existen las funciones anónimas, las cuales son asignadas (de ser necesario) a una variable que otorgará el nombre para poder ser invocada posteriormente:

```
const nombre = function (parametro1, parametro2, ...) {
    // ...
}
nombre();
```

Simulando el ejemplo anterior de la suma:

```
const suma = function(a, b) {
    return a + b;
}
```

Ejercicio guiado: Funciones anónimas

Solicitar al usuario que ingrese dos números enteros, y dentro de una función anónima se realice la división de ambos números, retornando y mostrando, en el mismo llamado de la función, el resultado como una variable. Para ello, sigamos los pasos:

- ***Paso 1*:** En el archivo funciones.js debes armar la estructura de la función anónima, es decir, sobre una variable llamada "divide" declaramos la función. Luego inicializar dos variables donde se almacenarán los datos ingresados por el usuario y serán pasados como argumento en el llamado a la función, en este caso, podemos llamar a las variables "num1" y "num2" y mediante la función "prompt()" solicitamos al usuario ingresar los datos. Este paso se vé de la siguiente forma:

```
var num1 = prompt("Ingrese el primer número: ");
var num2 = prompt("Ingrese el segundo número: ");

let divide = function (num1,num2) {
     //proceso
}
```

- ***Paso 2:*** Realizar el llamado a la función pasando los argumentos, en este caso, los dos números ingresados por el usuario y agregamos la función dentro de la estructura del mensaje con un document.write():

```
var num1 = prompt("Ingrese el primer número: ");
var num2 = prompt("Ingrese el segundo número: ");

let divide = function (a,b) {
    let resultado = parseInt(a) / parseInt(b);
    return resultado;
};

document.write("Resultado de la división: "+divide(num1,num2));
```

Generando el resultado en el navegador:

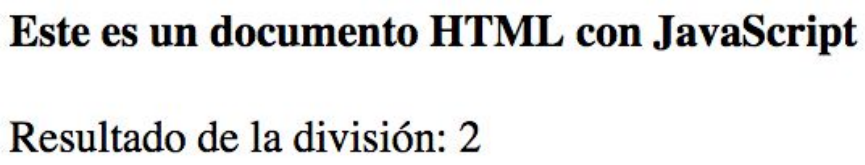

Imagen 2. Resultado de la función en el navegador.
Fuente: Desafío Latam

Alcance en una Función

El scope o alcance de una variable en JavaScript, se define como el espacio o segmentos de código donde esa función es conocida. Es decir, el scope decide a qué variables de tu programa tienes acceso directo. Sin embargo, la misma función delimita hasta donde conocen sus propias variables y hacia dónde hacen referencia estas mismas.

Ejercicio guiado: Alcance en una función

Demostrar cómo opera el alcance en las funciones, para esto: crear dos variables con el mismo nombre ubicadas en distintos espacios de código (dentro y fuera de la función) y mostraremos su valor. Para esto realizaremos los siguientes pasos:

- ***Paso 1:*** En el archivo funciones.js debemos agregar el código necesario para inicializar la variable “miVariable” con el número 10, luego crear una función llamada “`miFuncion`” que no recibirá ni retorna ningún tipo de valor, pero sí tendrá el inicio de la variable “miVariable” esta vez con el número 5 como valor inicial, luego dentro de

ella mostramos la variable recién creada dentro de la función con ayuda de un console.log().

```
var miVariable = 10; // variable global

function miFuncion () {
    var miVariable = 5 // se declara una "nueva" variable local al usar 'var'
    console.log(miVariable); // se muestra la variable en la consola
};
```

- ***Paso 2:*** Fuera de la función realizaremos un console.log() de "miVariable" y el llamado a la función "miFuncion". Por último, hacemos nuevamente el llamado mediante un console.log() de "miVariable" :

```
var miVariable = 10;  // variable global

function miFuncion () {
    var miVariable = 5 // se declara una "nueva" variable local al usar 'var'
    console.log(miVariable); // se muestra la variable en la consola
}

console.log(miVariable);
miFuncion(); // se hace el llamado de la función
console.log(miVariable);
```

Por ende, el resultado de la ejecución del código anterior sería:

```
10
5
10
```

Del resultado anterior podemos entender que el primer `10` corresponde a la declaración de `var miVariable = 10` que se encuentra al comienzo de nuestro código, el valor `5` corresponde al `console.log` de la función `miFuncion`, la cual en su interior declara otra variable con valor `5` y el último valor de `10` corresponde al valor de `miVariable` declarado al principio igual que el primer 10.

- ***Paso 3:*** Qué pasaría si en el código anterior, realizamos un pequeño cambio dentro de la función y no inicializamos la variable con la palabra reservada var y solamente le cargamos el número 5, como se muestra en el siguiente código:

```
var miVariable = 10; // variable global

function miFuncion () {
    miVariable = 5  // Se hace uso de la variable "global" 'miVariable'
    console.log(miVariable);
}

console.log(miVariable);
miFuncion();
console.log(miVariable);
```

El resultado de la ejecución para este caso sería:

```
10
5
5
```

En este ejemplo nos damos cuenta que el último `console.log` toma el valor de 5 declarado dentro de la función `miFuncion`, esto se debe a que dentro de esta función se está modificando el valor de la variable global `miVariable`, en palabras simples, en todo el código se está trabajando con la misma variable.

Ejercicio propuesto (3)

Desarrollar un programa en JavaScript donde se cree una variable global denominada "lenguaje" e inicie con el valor "JavaScript". Luego, crear una función denominada "miFuncion" y dentro de ella mostrar con un console.log el valor de la variable global. Posteriormente, crear otra variable dentro de la función denominada "lenguaje" e iniciar con el valor "NodeJS", seguidamente mostrar en un console.log la nueva variable creada.

Finalmente, hacer el llamado a la función y mostrar el valor de la variable global con un console.log.

Alcance de una Función dentro de una Función

Existe un concepto en informática llamado ***recursividad***, que en palabras simples consta de hacer referencia o definir un elemento en base a sí mismo. En las funciones, consiste en llamar nuevamente a la función desde ella misma de forma recursiva, de ahí el nombre, por lo tanto, se pueden hacer ciclos repetitivos mediante la recursividad de funciones y con la

ayuda de condicionales, para ir repitiendo una secuencia hasta que cierto valor llegue a un punto. Veamos un ejemplo donde se calcula el cuadrado de un número mediante recursividad.

```
function miFuncion() {
    let miVariable = 4;

    function potencia() {
      return miVariable * miVariable;
    }

    return potencia();
}

let resultado = miFuncion();
console.log(resultado);
resultado = potencia();
```

El resultado de la ejecución para **console.log(resultado)** sería:

```
16
```

Mientras que al tratar de asignar a la variable **resultado,** el resultado de la función **potencia** obtendremos:

```
Uncaught ReferenceError: potencia is not defined
```

Ejercicio guiado: Suma de términos

Calcular la suma de todos los términos almacenados en una variable con los datos del tipo arreglo del 1 al 9.

- ***Paso 1:*** En el archivo funciones.js debes armar la estructura de la función e inicializar una variable con datos del tipo arreglo, en este caso la podemos llamar “nums” e inicializarla con los valores del 0 al 9 dentro de un array: [0,1,2,3,4,5,6,7,8,9]. Quedando de la siguiente forma:

```
const nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function suma (nums) {
}

console.log(sum(nums))
```

- ***Paso 2:*** Como el dato se encuentra inmerso dentro de una sola variable del tipo arreglo, entonces la recursividad entre funciones es la solución para poder ir realizando la suma de todos los términos que se encuentran en el arreglo. Por lo tanto, dentro de la función, se debe crear una estructura de control de flujo con "if", para comprobar si ya no existen números dentro del arreglo para sumar, en el caso de no existir se debe retornar cero y detener la recursión, esto se puede lograr midiendo la longitud de la variable que contiene el arreglo con la instrucción "length".

```
const nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function suma (nums) {
    // La Recursión se detiene cuando un array está vacío
    if (nums.length < 1) { // con el length se mide la cantidad de
espacios que tiene la variable nums
        return 0;
    }
```

- ***Paso 3:*** Para poder finalizar el proceso en la función, se debe ir extrayendo y eliminando a la vez los números de la variable "nums" que contiene el arreglo de números. Esto se puede lograr mediante el método "shift()", el cual, se encarga de extraer y eliminar el primer término de un arreglo, sin importar el tipo de dato, por lo que se debe almacenar ese dato que extrae el método dentro de una variable "valor", para finalmente retornar la variable "valor" más el llamado de función principal pasando como argumento la variable "nums" que contiene los datos del tipo arreglo, quienes van a ir disminuyendo en cada recorrido de la función debido al método shift.

```
const nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function suma (nums) {
    // La Recursión se detiene cuando un array está vacío
    if (nums.length < 1) { // con el length se mide la cantidad de
espacios que tiene la variable nums
        return 0;
```

```
    }
    // El método shift() remueve el primer elemento del array
   // y retorna ese valor. Este método cambia la longitud del array
    var valor = nums.shift();

    // retornando en cada pasada la suma de los primeros valores que se
van removiendo
    return valor + suma(nums);
}
console.log(suma(nums))
```

Al revisar el resultado veremos:

```
45
```

Por lo tanto, en el ejercicio anterior se muestra como se puede llamar una función dentro de otra función logrando lo que se conoce como recursión. Para este caso en específico, la función sum retorna el valor que extrae del arreglo y le suma el resultado de la misma función, pero esta vez con un término menos en el arreglo debido al método shift(), que se encargará de extraer y eliminar cada número del arreglo hasta dejarlo sin ningún dato.

Ejercicio guiado: Factorial de un número

Desarrollar un código en JavaScript que permita calcular el factorial de un número. La fórmula para calcular el factorial es: **X! = X * (X-1)!**

Por ejemplo, si queremos calcular el factorial del número 4 (4!), sería igual a 4! = 1*2*3*4 = 24, por ende, se debe hacer el llamado a una función mientras el número enviado a la función siga siendo mayor que 1.

Llevemos a código todo lo planteado mediante el uso de las funciones, por lo tanto:

- ***Paso 1:*** En el archivo funciones.js, debemos agregar el código necesario para inicializar la variable que almacenará el dato ingresado por el usuario. Luego crear una función llamada "factorial_numero", que recibe esta información y retorna el resultado de la fórmula del factorial (x * (x-1)!), mientras x siga siendo mayor que 1. Esto en código se expresa de la siguiente manera:

```
function factorial_numero(x) {
      if (x > 1) {
                return x * factorial_numero(x - 1); // fórmula para el
factorial: X * (X-1)!
      }
```

- ***Paso 2:*** En el caso que x sea menor o igual a 1, debe retornar solamente el valor de x. Finalmente mostramos el resultado de la operación directamente desde el mensaje al final del programa.

```
function factorial_numero(x) {
      if (x > 1) {
                return x * factorial_numero(x - 1); // fórmula para el
factorial: X * (X-1)!
      } else {
                return x;
      }
}

var x = prompt("Ingrese un número");

document.write(`El factorial de ${x}! es: ${factorial_numero(x)}`);
```

Al ejecutar el código anterior, el resultado obtenido dependerá del valor ingresado por el usuario. Por ejemplo, si el usuario ingresa el número 4, el resultado sería:

```
El factorial de 4! es: 24
```

Si realizamos una corrida manual del código anterior, es decir, paso a paso a modo descriptivo y de pseudocódigo, podríamos analizar y observar lo que ocurre en cada paso:

```
Ve a factorial_numero(4).
¿4 es mayor a 1? Si. Coloca a factorial_numero(4) en espera y va a factorial_numero(3).
¿3 es mayor a 1? Si. Coloca a factorial_numero(3) en espera y va a factorial_numero(2).
¿2 es mayor a 1? Si. Coloca a factorial_numero(2) en espera y va a factorial_numero(1).
¿1 es mayor a 1? No. Evalúa factorial_numero(1).
Retoma factorial_numero(2).
Retoma factorial_numero(3).
Retoma factorial_numero(4).
```

Lo que sería igual a:

```
factorial_numero(4)
4 * factorial_numero(3)
4 * ( 3 * factorial_numero(2))
4 * ( 3 * ( 2 * factorial_numero(1)))
4 * ( 3 * ( 2 *  1  ))
4 * ( 3 *  2 )
4 * (6)
24
```

Ejercicio propuesto (4)

Realizar un programa con JavaScript que calcule la resta de todos los términos disponibles en una variable del tipo arreglo con los siguientes datos numéricos: nums = [1, 2, 3, 5, 8, 13, 21].

ECMAScript 6 (ES6)

Competencias

- Desarrollar una rutina utilizando variables locales y globales para controlar adecuadamente el alcance de la información en un programa.
- Desarrollar una función utilizando la notación de flecha de acuerdo al lenguaje ES6 para resolver un problema.

Introducción

La palabra ECMA es el acrónimo de *European Computer Manufacturers Association* y en el año 1997 se creó un comité para estandarizar JavaScript desde entonces ECMAScript es el estándar que nos indica cómo usar JavaScript. Hoy en día utilizamos la versión 6 (desde el 2015), pero la versión 5 estuvo tantos años que se convirtió en la versión más usada y tiene un alto porcentaje de compatibilidad, al principio no todos los sistemas y plataformas soportaban ES6 por lo que los transpiladores fueron de mucha ayuda.

Hoy en día gran parte de los sistemas y plataformas que existen soportan ES6. En el siguiente capítulo, aprenderás algunas de las características implementadas en ES6 como el alcance de las variables declaradas y las funciones flecha, que nos permiten tener un mejor control de nuestro código y optimizar la legibilidad del mismo

Alcance de variables declaradas let y var

Antes de ES6 sólo se podían declarar variable usando `var` las cuales al momento de ser usadas dejaban a nuestras variables disponibles fuera del contexto o bloque donde son declaradas, debido a que `var` tiene ambiente de función, esto significa que cuando se ejecuta el código y se detecta una variable `var`, JavaScript la "eleva" (hoisting), ¿qué quiere decir esto? mejor veámoslo con un ejemplo.

Si tenemos el siguiente bloque de código:

```
for(var i = 0; i < 5; i++) {
  console.log(i);
}
```

El interpretador de JavaScript al encontrarse con `var i = 0` dentro del for la eleva dejando el código así:

```
var i;
for(i = 0; i < 5; i++) {
  console.log(i);
}
```

Por esta razón sucede lo que veremos a continuación:

```
for(var i = 0; i < 5; i++) {
  console.log(i);
}
console.log(i + 2);
```

Veamos el resultado entregado:

```
0
1
2
3
4

7
```

Si observas el último `console.log` nos retornó 7, lo que significa que la variable `i` sigue disponible fuera del contexto del ciclo for donde fue declarada.

Veamos qué pasa entonces cuando ejecutamos un ciclo for anidado. ¿Qué pasa con los valores de la variable `i`?:

```
for (var i = 0; i <= 2; i++) {
    console.log("Ciclo externo " + i);
    for (var i = 0; i <= 2; i++) {
      console.log("Ciclo interno " + i);
  }
}
```

Al ejecutar este código esperaríamos la siguiente respuesta:

```
Ciclo externo 0
Ciclo interno 0
Ciclo externo 1
Ciclo interno 1
Ciclo externo 2
Ciclo interno 2
```

Pero la realidad es esta:

```
Ciclo externo 0
Ciclo interno 0
Ciclo interno 1
Ciclo interno 2
```

Esto sucede porque la variable `i` al terminar el ciclo interior con **i=2** se mantiene con el mismo valor en la siguiente iteración del ciclo externo.

Con ES6 nace la definición de variables con la palabra protegida **let** que permite definir variables que solo existen dentro del contexto o bloque donde se declaró. Revisemos los mismos ejemplos anteriores pero usando **let** en vez de **var** y veamos la diferencia en sus resultados.

```
for(let i = 0; i < 5; i++) {
  console.log(i);
}
```

```
console.log(i + 2);
```

Veamos el resultado usando let:

```
0
1
2
3
4

ReferenceError: i is not defined
```

Si observas bien, ahora el `console.log` fuera del ciclo **for** indica que "la variable i no se ha definido" y esto es por que sólo existe dentro del contexto del ciclo **for** y no fuera de él**.** Veamos otro ejemplo implementando la variable de tipo let con un ciclo for anidado (uno dentro de otro), en donde se muestra mediante un "console.log()" la variable declarada en el primer ciclo for como ciclo externo y luego mediante un "console.log()" la variable i declarada en el for como ciclo interno .

```
for (let i = 0; i <= 2; i++) {
    console.log("Ciclo externo " + i);
    for (let i = 0; i <= 2; i++) {
      console.log("Ciclo interno " + i);
  }
}
```

Al utilizar **let,** el valor se mantiene dentro del contexto donde se encuentra. Por lo tanto, al revisar el resultado del código anterior, encontraríamos lo siguiente:

```
Ciclo externo 0
Ciclo interno 0
Ciclo interno 1
Ciclo interno 2
Ciclo externo 1
Ciclo interno 0
Ciclo interno 1
Ciclo interno 2
Ciclo externo 2
Ciclo interno 0
Ciclo interno 1
Ciclo interno 2
```

Como se puede observar en el resultado, la variable i del primer ciclo for no interviene en la variable i del segundo ciclo for, por esta razón, el implementar variables con let nos garantiza que dichas variables sólo serán válidas en su contexto y no tendrán injerencia en otro contexto, como por ejemplo, otro ciclo anidado.

Veamos si quedó claro e intenta predecir el resultado de los `console.log()`.

```
function probando() {
  var i = 10;
  if(true) { // siempre entrará dentro del if por ser true su condición
    let i = 30;
    console.log(i)
  }
  console.log(i);
}
probando()
```

¿Cuál crees que es la respuesta y por qué? Veamos si respondiste correctamente, para eso antes revisemos el resultado:

```
30
10
```

El primer `console.log()` corresponde al `let i = 30` y el segundo corresponde al `var i = 10`. Los valores definidos se respetaron ya que el `i` dentro del if fue declarado como **let,** entonces sólo existe dentro de ese contexto y no sobrescribe la variable ya definida, por más que esta haya sido declarado como **var**.

Usar **let** o **var** depende de cada situación y problema a resolver ya que en algunos casos no importará si la variable se mantiene fuera del bloque, pero hay otros en que sí y en ese caso debemos usar **let** para definir.

Ejercicio guiado: Alcance de variables

Desarrollar un código en JavaScript que permita calcular la suma y la resta de dos números por funciones separadas, pero manteniendo el mismo nombre de las variables en ambas operaciones.
Además, los datos ingresados por el usuario se deben guardar en variables separadas dentro de cada función con el mismo nombre para las variables. Sigamos los siguientes pasos:

- ***Paso 1*:** En el archivo funciones.js, se debe armar la estructura de las funciones, una para la suma y otra para la resta, ambas funciones no van a recibir ningún tipo de datos, por lo tanto, en la función para la suma se crearán dos variables con "let" (num1 y num2) para guardar los números solicitados al usuario (variables locales), así mismo se realizará la suma de ambos números y se retornará con el resultado. La variable para almacenar el resultado de la suma y resta debe ser inicializada fuera de las funciones con var (siendo una variable global), quedando la función de suma de la siguiente forma:

```
function suma() {
      let num1 = prompt("Ingrese un primer número para al suma");
      let num2 = prompt("Ingrese un segundo número para al suma");
      resultado = parseInt(num1) + parseInt(num2);
      return resultado;
};
```

- ***Paso 2:*** Realizada la función de la suma, pasemos a realizar la función de la resta la cual es muy parecida, lo único que se debe considerar es que esta vez se restan los dos valores ingresados por el usuario, como se muestra a continuación:

```
function resta() {
      let num1 = prompt("Ingrese un primer número para al resta");
      let num2 = prompt("Ingrese un segundo número para al resta");
      resultado = parseInt(num1) - parseInt(num2);
      return resultado;
};
```

- ***Paso 3:*** Para finalizar el ejercicio, solo se debe crear la variable para el resultado de manera global y hacer el respectivo llamado a ambas funciones:

```
function suma() {
      let num1 = prompt("Ingrese un primer número para al suma");
      let num2 = prompt("Ingrese un segundo número para al suma");
      resultado = parseInt(num1) + parseInt(num2);
      return resultado;
};

function resta() {
      let num1 = prompt("Ingrese un primer número para al resta");
      let num2 = prompt("Ingrese un segundo número para al resta");
```

```
      resultado = parseInt(num1) - parseInt(num2);
      return resultado;
};
var resultado;
document.write(`El resultado de la suma es: ${suma()} <br>`);
document.write(`El resultado de la resta es: ${resta()}`);
```

Al ejecutar el código anterior, el resultado obtenido dependerá del valor ingresado por el usuario. Por ejemplo, si el usuario ingresa los números 4 y 6 para la suma, mientras que para la resta ingresa 5 y 11 el resultado sería:

```
El resultado de la suma es: 10
El resultado de la resta es: -6
```

Por ende, se puede apreciar que hemos utilizado la variable "resultado" con un "var" fuera de la función y la podemos referenciar dentro de las funciones sin la necesidad de volver a declararla, pero en cambio, las variables num1 y num2 dentro de las funciones sí se deben volver a declarar.

Ejercicio propuesto (5)

Lee los siguientes códigos, y sin ejecutarlo, interpreta ¿cuál es el valor de `i` al ejecutarse la sentencia console.log(i) al final del código?

Luego, ejecuta el código para comparar tus resultados.

```
let i = 0;
while(i < 10) {
  i++;
}
console.log(i)
```

```
function recorrer() {
    let i = 0;
    while(i < 10) {
    i++;
  }
}
recorrer();
console.log(i)
```

Función Arrow

Con ES6 también nacen las arrow functions que es una forma simplificada de escribir o definir una función, incluso el poder tener una función en una sola línea de código. Veamos unos ejemplos:

Hasta el momento hemos definido funciones de la siguiente manera:

```
function miFuncionSumar(a,b) {
   return a + b;
}

console.log(miFuncionSumar(1,2))
```

Resultado:

```
3
```

Ahora con ES6 podemos escribir funciones eliminando la palabra **function** y usando **=>**, esto transformaría la función del ejemplo anterior en:

```
let miFuncionSumar = (a,b) => {
   return a + b;
}

console.log(miFuncionSumar(1,2));
```

Resultado:

```
3
```

Para practicar, escribe una función de la forma tradicional y luego conviértela a una arrow function. Por ejemplo, la función anterior denominada `miFuncionSumar` la podemos reescribir de la siguiente manera, implementando ES6 arrow functions:

```
let miFuncionSumar = (a,b) => a + b;
```

El ejemplo anterior es similar a:

```
let miFuncionSumar = (a,b) => { a + b };
```

Pero al ser una función de una sola sentencia podemos omitir las { }.

Para profundizar el concepto de arrow functions, puedes consultar el documento **Material Apoyo Lectura - Funciones Flecha,** ubicado en "Material Complementario". En este documento podrás revisar en detalle aspectos teóricos y prácticos de este contenido.

Ejercicio guiado: Arrow Functions

En el siguiente código, se solicita al usuario ingresar un número entero para enviarlo como argumento en el llamado a la función y recibirlo como parámetro para poder comparar el valor mediante una estructura condicional.
Realizar la transformación a funciones flecha o arrow de ES6

```
var num = prompt("Ingrese un numero entero");
var resultado = 0;

function verificar(numero) {
      if (numero > 0) {
            resultado = "positivo";
      } else if (numero < 0) {
            resultado = "negativo";
      } else if (numero === 0) {
            resultado = "nulo";
      } else {
            resultado = "no es un número";
      }
      return resultado
}

alert(`El numero ingresado es: ${verificar(parseInt(num))}`);
```

- ***Paso 1*:** Para transformar el código anterior a funciones arrow de ES6, lo primero que debemos observar es la estructura actual, en este caso, la función tiene parámetros y distintos procesos dentro de ella, por lo tanto, se deben utilizar los paréntesis "(parametro1...)" y las llaves "{proceso}" para separar los procesos dentro de la función original, agregando la flecha después del parámetro:

```
var num = prompt("Ingrese un numero entero");
var resultado = 0;

let verificar = (numero) => {  // cuerpo de la función donde debe ir
todo el proceso
}
```

- ***Paso 2:*** Agregamos toda la estructura condicional if-else-if dentro de la función, almacenando el mensaje correspondiente en la variable global y retornando el resultado dentro de esa variable:

```
var num = prompt("Ingrese un numero entero");
var resultado = 0;

let verificar = (numero) => {
      if (numero > 0) {
            resultado = "positivo";
      } else if (numero < 0) {
            resultado = "negativo";
      } else if (numero === 0) {
            resultado = "nulo";
      } else {
            resultado = "no es un número";
      }
      return resultado
}

alert(`El numero ingresado es: ${verificar(parseInt(num))}`);
```

Siendo el resultado para el valor -5:

```
El numero ingresado es: negativo
```

Como se puede observar en el código anterior, la mayor modificación está en la inicialización de la función. Quitando la palabra reservada “function” y agregando la flecha correspondiente ”=>”. En el caso del return, cuando el cuerpo de la función es más de una línea, es decir, se trabaja con las llaves, el return debe ir dentro de la función obligatoriamente si se desea retornar algún valor.

Ejercicio propuesto (6)

Realizar un programa en JavaScript mediante el uso de funciones flecha que solicite al usuario ingresar su nombre, edad y retorne un saludo con el nombre, la edad e indicando si es mayor o menor de edad.

Comparativa entre ES5 y ES6

A continuación, veremos una tabla con las principales diferencias entre la versión ES5 y ES6.

Característica	ES5	ES6
Retornar un número	`function retornaNum() { return 3 }`	`let retonarNum = () => 3;`
Retornar un número de una función con parámetros	`function sumarDos(num) { return num + 2; }`	`let sumarDos = (num) => num + 2;`
Alcance de bloque (let)	No existe.	`let numero = 1;`
Template string	`let miNombre = "Jon Doe" let cadena = "Hola " + miNombre;`	`let miNombre = "Jon Doe" let cadena = `Hola ${miNombre}`;`

Tabla 1. Comparativa entre ES5 y ES6
Fuente: Desafío Latam

Cuando estamos desarrollando, es importante conocer si los navegadores soportan la versión del lenguaje que estamos utilizando, sobretodo si es nueva. Para conocer cómo validar esta información y profundizar tu conocimiento sobre compatibilidad, puedes consultar el documento **Material Apoyo Lectura - Transpiladores y compatibilidad,** ubicado en “Material Complementario”.

Ciclos

Competencias

- Reconocer los aspectos clave de los ciclos y estructuras de control para construir algoritmos que requieran la utilización de ciclos anidados.
- Desarrollar algoritmos utilizando ciclos de instrucciones if/else y ciclos anidados, para resolver un problema de baja complejidad

Introducción

Los lenguajes de programación en general nos permiten realizar tareas de cualquier naturaleza de manera rápida, esto es bastante beneficioso cuando la tarea a realizar es de carácter repetitivo, es decir, una misma tarea ejecutándose múltiples veces bajo parámetros ligeramente diferentes.

En este capítulo abordaremos y le daremos más profundidad a las estructuras para realizar ciclos. Manejar estos conceptos te permitirá optimizar la forma en que escribes código, entregándote herramientas para resolver algoritmos cada vez más complejos como los que te encontrarás en el mundo laboral.

Rutinas Repetitivas v/s Ciclos

Los ciclos en programación nos permiten llevar a cabo tareas que son repetitivas, como por ejemplo, calcular el promedio de notas para cada estudiante. En un curso de 10 estudiantes la operación para ello es siempre la misma: promediar y/o ponderar las notas totales y generar una nota final. Este proceso se repite para cada uno de los estudiantes.

Consideremos que las notas de los estudiantes están guardadas dentro de un Array, luego el código en JavaScript se vería algo así (siendo **calcularPromedio** la función para calcula el promedio de un solo estudiante):

```
calcularPromedio(estudiante[0]);
calcularPromedio(estudiante[1]);
calcularPromedio(estudiante[2]);
calcularPromedio(estudiante[3]);
calcularPromedio(estudiante[4]);
calcularPromedio(estudiante[5]);
calcularPromedio(estudiante[6]);
calcularPromedio(estudiante[7]);
calcularPromedio(estudiante[8]);
calcularPromedio(estudiante[9]);
```

Si tomamos en cuenta que el proceso para calcular el promedio es el mismo para todos los estudiantes, consideramos que es una tarea que se repite sin mayores cambios entre un estudiante y el siguiente, entonces podemos “iterar” entre los estudiantes para realizar esta operación:

```
ciclo(estudiante = 1; estudiante = 10; estudiante++) {
    calcularPromedio(estudiante)
}
```

Un ciclo en programación suele tener 3 elementos para realizar su ejecución:

Inicializador	Usualmente es una variable asignada a un número para definir un parámetro de inicio del ciclo, suele ser conocido también como contador.
Condición de salida/término	Corresponde a un Boolean, que define si se debe continuar con el ciclo o se debe detener.
Expresión final	Corresponde a una expresión u operación que se ejecuta al final del bloque del ciclo, suele ser una operación que acerca el contador a la condición de salida.

Tabla 2. Elementos de un ciclo
Fuente: Desafío Latam

Diagrama de Flujo

Ya mencionados los 3 elementos básicos para realizar un ciclo, es posible hacer su símil con los componentes vistos con anterioridad:

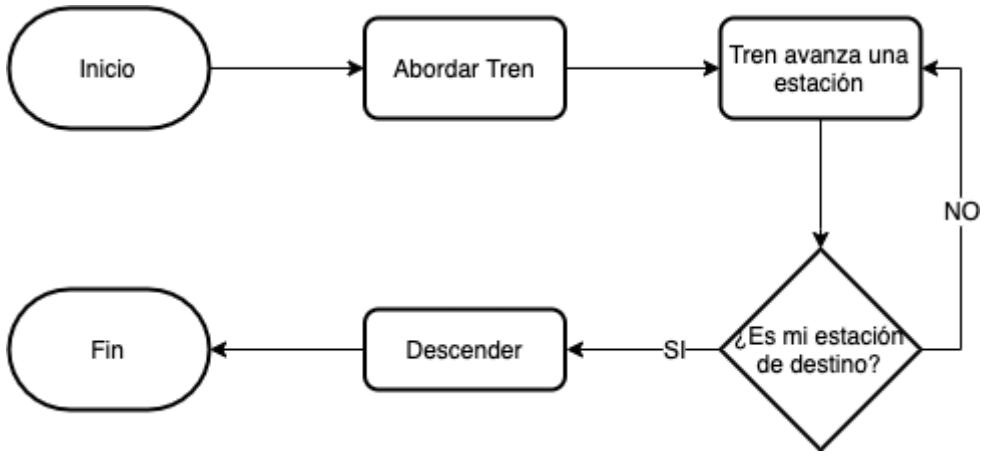

Imagen 3. Diagrama de flujo para un viaje en tren.
Fuente: Desafío Latam

Inicializador: Es un proceso que consta de asignar las variables a su punto inicial, en este caso particular "Abordar Tren" o bien, estación.

Expresión final: Proceso de incrementar (para el ejemplo) "Tren avanza una estación".

Condición de salida/término: Rombo de decisión, en este caso las salida se evalúa mediante la pregunta "¿Es mi estación de destino?".

Ciclos y Estructuras de Control

Existen diferentes tipos de estructuras de control, para en este capítulo nos enfocaremos en las relacionadas con ciclos. Ya conocemos la estructura básica **for,** con sus 3 componentes los cuales sirven para definir los parámetros de iteración de ciclo. Sin embargo, existen otras 2 palabras reservadas para alterar el funcionamiento "normal" de un ciclo: **break** y **continue**.

La palabra **break** en español significa quebrar y efectivamente eso es lo que logra en un ciclo, desde el punto (línea) donde se escribe esta palabra, el ciclo se interrumpe, quiebra o finaliza.

```
for (let i = 1; i < 10; i++) {
    if (i == 3) {
      break;
    }
    console.log(i);
}
```

El resultado en la consola del navegador será el siguiente:

```
1
2
```

Para el caso del ejemplo, al cumplirse la condición **i == 3** y llegar a la sentencia **break**, **for** finaliza su ejecución, por ende, el ciclo concluye aunque la condición de término aún se cumpla **(i < 10)**.

De manera análoga **continue** (continuar en español), nos permite ignorar los comandos siguientes en el bloque de código del ciclo sin terminar la iteración completa sino solo el segmento actual.

```
for (let i = 1; i < 10; i++) {
    if (i == 3) {
      continue;  // se ignora la sentencia 'console.log()'
    }
    console.log(i);
}
```

El resultado en la consola del navegador será el siguiente:

```
1
2
4
5
6
7
8
9
```

Ejercicio guiado: Estructuras de control y repetitivas

Desarrollar un ejercicio utilizando estructuras de control y repetitivas, lo cual nos permitirá identificar y diferenciar entre ambas estructuras, así como el funcionamiento y utilización de cada una. El ejercicio consiste en mostrar los números pares y contar los números impares mediante un ciclo for para los números comprendidos entre el 0 y el 20 (incluidos ambos), pero cuando se llegue al número 10, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 19 el ciclo debe ser interrumpido y terminar su ejecución. Para ello, sigamos los siguientes pasos:

- ***Paso 1*:** En el archivo script.js debes armar la estructura del ciclo repetitivo "for" para que pueda realizar el conteo desde el número 0 hasta el número 20. Quedando de la siguiente forma:

```
for (let i = 0; i <= 20; i++) {
    //cuerpo del ciclo repetitivo
}
```

- ***Paso 2*:** Teniendo listo el ciclo for, proseguimos dando solución al planteamiento del problema, en este caso se pide mostrar los números pares y contar los números impares, además, no mostrar el número 10 y terminar el ciclo cuando se llegue al número 19. Esto se puede lograr mediante el uso de estructuras de control if-else, verificando si el número es par (mostrar) de lo contrario impar (contar), o es 10 o es 19 para ejecutar la acción correspondiente, como por ejemplo, en el caso de ser 10 no se debe mostrar el número, utilizando la sentencia "continue" logramos este objetivo. Para terminar la ejecución del ciclo cuando se esté en el número 19, con la sentencia "break" se puede detener abruptamente el ciclo. Llevando todo esto al código encontraríamos:

```
var impar = 0;

for (let i = 0; i <= 20; i++) {
    if (i == 10) {
      continue;  // se ignora cualquier otro proceso
    };
    if (i == 19) {
      break;  // se rompe el ciclo actual
    };
    if (i % 2 == 0) {
      document.write(i+"<br>");
    }else {
      impar++;
    };
};
document.write("La cantidad de números impares es: "+impar);
```

Al ejecutar el código anterior, el resultado mostrado en nuestro navegador sería:

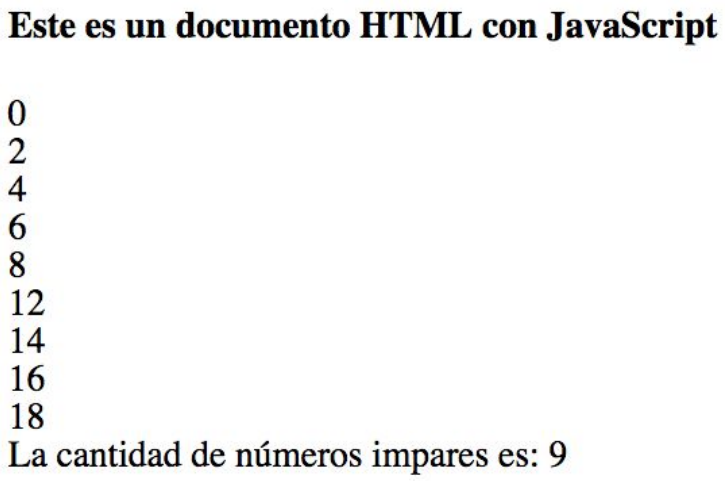

Imagen 4. Resultado final del ejercicio en el navegador.
Fuente: Desafío Latam

Ejercicio propuesto (7)

Mostrar los números impares y contar los números pares mediante un ciclo for para los números comprendidos entre el 1 y el 50 (incluidos ambos), pero cuando se llegue al número 11, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 40 el ciclo debe ser interrumpido y terminar su ejecución.

while y do...while

Como ya hemos visto, los bucles son útiles si se desea ejecutar el mismo código una y otra vez con un valor diferente. Ejemplo de esto son los ciclos while y do-while, que se traduce al español como "mientras" y "hacer mientras" se cumpla una condición, tienen la siguiente estructura:

```
inicializador;

while (condición) {
  // ...
  expresión final
}
```

```
inicializador;

do {
  // ...
  expresión final
} while (condición);
```

Debido a que no son parte integral de la estructura **while/do...while** el inicializador y la expresión final no tienen por qué utilizar esos lugares y pueden incluso estar ausentes. Es necesario tener en cuenta que la ausencia de alguno de estos elementos puede causar diversos problemas, como una iteración infinita o la no ejecución del bloque si la condición no se cumple, desde un inicio con el caso del **while**. Recuerda que la diferencia entre **while** y **do...while** es que este último se ejecuta al menos 1 vez ya que la condición se evalúa al final del bloque y no antes de como en el **while**.

```
while (1) {
    console.log("Saludos");
}
```

```
do {
    console.log("Saludos");
} while (1);
```

¿Sabes qué hace el código anterior? Cómo es posible observar, no posee inicializador ni expresión final. Además, la condición es de tipo Number diferente de 0, por ende, su conversión a Boolean es true. Luego, el ciclo se ejecutará infinitas veces.

Otro error bastante común que producen los ciclos infinitos en la creación de ciclos con while es olvidar ingresar una expresión final. En el caso del siguiente ejemplo, el valor de **i** nunca se incrementa, en consecuencia, siempre será menor a 10. Lo cual generaría un ciclo infinito en nuestro navegador web al ejecutar el programa.

```
let i = 0;
while(i < 10) {
  console.log(i);
}
```

```
let i = 0;
do {
  console.log(i);
} while (i < 10);
```

Otro aspecto importante a considerar, es que debes definir en algún momento un inicializador para hacer la condición válida. Todos los puntos anteriores son recomendaciones y no son "exigidas" por JavaScript. Esto es porque con las herramientas vistas, es posible emular un ciclo normal, por ejemplo, si se desea solicitar indefinidamente a un usuario que ingrese un número hasta que el usuario ingrese un número que sea mayor o igual a 10, debemos implementar el break para romper el ciclo y poder salir de él, como se muestra a continuación:

```
let ingreso;
while (true) {
  ingreso = prompt("Ingrese un número: ");
  if (ingreso >= 10) {
    break;
  }
}
```

Ejercicio guiado: Números impares

Se solicita mostrar los números pares y contar los números impares mediante un ciclo while para los números comprendidos entre el 2 y el 15 (incluidos ambos), pero cuando se llegue al número 9, el programa debe continuar y no mostrar el número en cuestión. Igualmente, cuando se llegue al número 14 el ciclo debe ser interrumpido y terminar su ejecución.

- ***Paso 1*:** En el archivo script.js debes armar la estructura del ciclo repetitivo "while" para que pueda realizar el conteo desde el número 2 hasta el número 15. Por lo que

debemos declarar e iniciar una variable fuera del ciclo con el valor por defecto (1) y dar la condición al ciclo while para que termine cuando llegue al número 15. Quedando de la siguiente forma:

```
let impar = 0;
let i = 1;

while(i <= 15) {
    //cuerpo de la estructura repetitiva while
}
```

- ***Paso 2***: Teniendo listo el ciclo while, proseguimos dando solución al planteamiento del problema, en este caso se pide mostrar los números pares y contar los números impares, además, no mostrar el número 9 y terminar el ciclo cuando se llegue al número 14. Esto se puede lograr mediante el uso de estructuras de control if-else, verificando si el número es par (mostrar número) de lo contrario impar (contar número), o es 9 o es 14 para ejecutar la acción correspondiente, como por ejemplo, en el caso de ser 9 no se debe mostrar el número, utilizando la sentencia "continue" logramos este objetivo, y para terminar la ejecución del ciclo cuando se esté en el número 14, con la sentencia "break" se puede detener abruptamente el ciclo. Llevando todo esto al código encontraríamos:

```
let impar = 0;
let i = 1;

while(i <= 15) {
    i++;
    if (i == 9) {
      continue;  // se ignora cualquier otro proceso
    };
    if (i == 14) {
      break;  // se rompe el ciclo actual
    };
    if (i % 2 == 0) {
      document.write(i+"<br>");
    }else {
      impar++;
    };
};
document.write("La cantidad de números impares es: "+impar);
```

Al ejecutar el código anterior, el resultado en nuestro navegador sería:

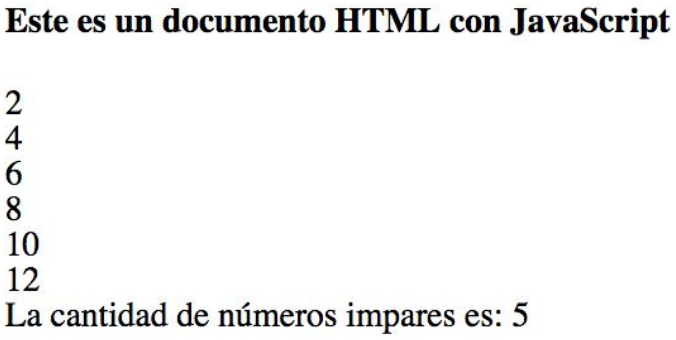

Imagen 5. Resultado final del ejercicio en el navegador.
Fuente: Desafío Latam

Ejercicio guiado: Múltiplos de cuatro

Se solicita mostrar los números múltiplos de cuatro y contar los números que no sean múltiplo de cuatro mediante un ciclo do-while para los números comprendidos entre el 1 y el 50 (incluidos ambos), pero cuando se llegue al número 16, el programa debe continuar y no mostrar el número en cuestión. Igualmente, cuando se llegue al número 45 el ciclo debe ser interrumpido y terminar su ejecución. Para ello realiza los siguientes pasos:

- ***Paso 1*:** En el archivo script.js debes armar la estructura del ciclo repetitivo "do-while" para que pueda realizar el conteo desde el número 1 hasta el número 50. Por lo que debemos declarar e iniciar una variable fuera del ciclo con el valor por defecto (1) y dar la condición al ciclo do-while para que termine cuando llegue al número 50. Quedando de la siguiente forma:

```
let no_multiplo = 0;
let i = 1;

do{
    //cuerpo de la estructura repetitiva do-while
}while(i <= 50)
```

- ***Paso 2:*** Teniendo listo el ciclo do-while, proseguimos dando solución al planteamiento del problema, en este caso se pide mostrar los números múltiplos de cuatro y contar los que no son múltiplos de cuatro, además, no mostrar el número 16 y terminar el ciclo cuando se llegue al número 45. Esto se puede lograr mediante el uso de estructuras de control if-else, verificando si el número es múltiplo de cuatro (mostrar número) de lo contrario no es múltiplo de cuatro (contar número). Llevando todo esto al código encontraríamos:

```
let no_multiplo = 0;
let i = 1;

do {
    if (i == 16) {
      i++;
      continue;  // se ignora cualquier otro proceso
    };
    if (i == 45) {
      break;  // se rompe el ciclo actual
    };
    if (i % 4 == 0) {
      document.write(i+"<br>");
    }else {
      no_multiplo++;
    };
    i++;
}while(i <= 50);
document.write("La cantidad de números no múltiplos de cuatro es:
"+no_multiplo);
```

Al ejecutar el código anterior, el resultado en nuestro navegador sería:

Este es un documento HTML con JavaScript

4
8
12
20
24
28
32
36
40
44
La cantidad de números no múltiplos de cuatro es: 33

Imagen 6. Resultado final del ejercicio en el navegador.
Fuente: Desafío Latam

Ejercicio propuesto (8)

Realizar un programa con JavaScript mediante el ciclo while que muestre los números múltiplos de cinco y cuente los números que no sean múltiplo de cinco, ara los números comprendidos entre el 0 y el 35 (incluidos ambos). Cuando se llegue al número 10, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 30 el ciclo debe ser interrumpido y terminar su ejecución.

Ejercicio propuesto (9)

Realizar un programa con JavaScript mediante el ciclo do-while que muestre los números múltiplos de seis y cuente los números que no sean múltiplo de seis, para los números comprendidos entre el 2 y el 63 (incluidos ambos). Cuando se llegue al número 19, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 59 el ciclo debe ser interrumpido y terminar su ejecución.

Para continuar ejercitando y entender mejor cómo resolver problemas aplicando ciclos, puedes consultar el documento **Material Apoyo Lectura - Programación dentro de un ciclo,** ubicado en “Material Complementario”. En este documento podrás ver el paso a paso, cómo abordar problemas complejos.

Ciclos anidados

Se conocen como ciclos anidados cuando la declaración de un ciclo es otro ciclo, generalmente usamos ciclos anidados cuando usamos operaciones un poco más complejas, veamos el siguiente ejemplo.

Busquemos todas las combinaciones entre el elemento a y b:

```
for(let i = 0; i < 3; i++) {
   for(let j = 0; j < 3; j++) {
        console.log(i,j);
    }
}
```

Resultado

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Como te diste cuenta en el ejemplo anterior el **for** interno se repitió 3 veces más que fue la cantidad de iteración que realizó el **for** externo.

Veamos otro ejemplo y calculemos los factoriales del 1 al 10. Anteriormente realizamos este ejercicio pero implementando recursividad de funciones, ahora lo realizaremos a través de anidación de ciclos para poder obtener los factoriales del 1 al 10, si no recuerdas cómo obtener el factorial de un número básicamente es multiplicar desde 1 hasta el número, por ejemplo obtengamos el número factorial de 5 (el factorial se representa con un !).

```
5! = 1 * 2 * 3 * 4 * 5
```

El resultado de la multiplicación sería **120**, entonces 5! = 120.

Ejercicio guiado: Factoriales

Desarrollar un programa utilizando JavaScript que permita encontrar el factorial del 1 al 10 implementando anidación de ciclos repetitivos. Sigamos los pasos a continuación:

- ***Paso 1*:** Separar el problema en partes. Lo primero que haremos será calcular el factorial de un número, analizando la fórmula podemos darnos cuenta lo más apropiado es usar un for para resolver el problema.

```
var res = 1;
for (let i = 1; i <= 5; i++) {
    res = res * i;
}
console.log(res);
```

Al ejecutar el código obtenemos:

```
120
```

- ***Paso 2*:** Hemos generado el código para calcular el número factorial de 5, pero nosotros necesitamos obtener los primeros 10 factoriales y para eso necesitamos repetir el código anterior 10 veces, así que utilizaremos un ciclo dentro de otro.

```
for (let i = 1; i <= 10; i++) {
   let res = 1;
   for (let j= 1; j <= i; j++) {
      res = res * j;
  }
  console.log("Factorial de " + i + " es: " + res);
}
```

Al ejecutar el código anterior, obtenemos los factoriales para los números desde el 1 hasta el 10 mediante la anidación de ciclos repetitivos:

```
Factorial de 1 es: 1
Factorial de 2 es: 2
Factorial de 3 es: 6
Factorial de 4 es: 24
Factorial de 5 es: 120
Factorial de 6 es: 720
Factorial de 7 es: 5040
Factorial de 8 es: 40320
Factorial de 9 es: 362880
Factorial de 10 es: 3628800
```

Este código y seguramente muchos otros que desarrollemos en el futuro, son posible de reestructurarse para mejorar su legibilidad. Para conocer cómo realizar esto paso a paso, puedes consultar el documento **Material Apoyo Lectura - Transformar ciclos en funciones,** ubicado en “Material Complementario”.

Ejercicio guiado: Elementos comunes en dos listados

Desarrollar un programa utilizando JavaScript que permita encontrar los elementos comunes entre dos listados de números realizados con ciclos repetitivos anidados, uno va del 1 al 5 y el segundo del 1 al 10. Por consiguiente:

- ***Paso 1*:** Como se desea mostrar los números que sean iguales entre dos listas generadas por ciclos repetitivos anidados, entonces vamos a crear dos ciclos for anidados, uno del 1 al 10 y el otro del 1 al 20 y así mostrar ambas variables utilizadas para el conteo de cada ciclo, como se muestra a continuación:

```
for(let i = 0; i < 5; i++) {
     console.log("i: "+i);
   for(let j = 0; j < 10; j++) {
     console.log("j: "+j);
   }
}
```

- ***Paso 2*:** Ejecutamos el código anterior y obtendremos lo siguiente:

```
i=0
j=1
j=2
j=3
…
…
…
j=9
i=2
j=0
j=1
…
```

- ***Paso 3:*** Como se puede apreciar en la ejecución anterior, primero se muestran el valor del contador "i" y luego todos los valores del contador "j", repitiendo el ciclo hasta llegar a la última iteración del contador "i" por ser el contador externo. Ahora si debemos buscar los elementos que sean iguales entre ambas listas, para ello basta con preguntar mediante una estructura repetitiva si las variables de cada ciclo for son igual (ciclos repetitivos for anidados), quedando de la siguiente forma:

```
for(let i = 0; i < 5; i++) {
   for(let j = 0; j < 10; j++) {
       if(i == j) {
            console.log("El número "+ i +" se encuentra en ambos
listados")
       }
    }
}
```

- ***Paso 4:*** *E*jecutamos el programa y así se puede observar el resultado del código anterior:

```
El número 0 se encuentra en ambos listados
El número 1 se encuentra en ambos listados
El número 2 se encuentra en ambos listados
El número 3 se encuentra en ambos listados
El número 4 se encuentra en ambos listados
```

Ejercicio guiado: Encontrar elementos en un arreglo

Desarrollar un programa utilizando JavaScript partiendo del nombre específico de un estudiante, Juan. Se solicita verificar mediante un arreglo si este estudiante se encuentra en el curso de Diseño donde existe una lista de 5 estudiantes: Pedro, María, Diego, Juan y Paola.

- ***Paso 1*:** En el archivo script.js, inicializamos las variables para el nombre del estudiante que se desea buscar y para almacenar la lista de estudiantes del curso, por lo que quedaría:

```
let estudiante = "Juan"
let cursoDiseno = ["Pedro","Maria","Diego","Juan","Paola"]
```

- ***Paso 2:*** En este ejercicio se solicita verificar si "Juan" pertenece al curso de Diseño, por lo que deberíamos ver si “Juan” está en alguna de las posiciones del array. Para esto, se puede implementar un ciclo con for para hacer tantas iteraciones como nombres existan en la lista de alumnos, luego dentro del ciclo for utilizando estructuras condicionales como el if, comprobaremos si el estudiante se encuentra dentro de esa lista, accediendo a cada una de las posiciones del arreglo y preguntando si son iguales los nombres. Para acceder a una posición específica en una variable que contiene un arreglo se implementan los corchetes después del nombre de la variable. En el caso de encontrarse, modificamos el estado de una variable con dato booleano de falso a verdadero.

```
let loEncontramos = false;
for (let i = 0; i < cursoDiseno.length; i++) {
    if(cursoDiseno[i] ===  estudiante) {
      loEncontramos = true;
    };
};
```

- ***Paso 3*:** Por último, solo queda preguntar por el estado de la variable modificada dentro del ciclo for, esta variable por defecto se encuentra en false, solo cambia a true si se encuentra el nombre en la lista, de lo contrario siempre será false, eso se puede aprovechar para utilizar la estructura condicional if.

```
let loEncontramos = false;
for (let i = 0; i < cursoDiseno.length; i++) {
    if(cursoDiseno[i] ===  estudiante) {
      loEncontramos = true
    }
}

if(loEncontramos) {
  console.log('Juan pertenece al curso de diseño')
}
```

El código anterior revisa si alguno de los valores del array es "Juan" y si lo encuentra lo guarda para luego mostrar el mensaje, que debería dar el siguiente resultado:

```
Juan pertenece al curso de diseño
```

- ***Paso 4:*** Ahora imagina que en vez de buscar un estudiante te piden revisar si 2 estudiantes (Juan y Maria) pertenecen al curso de Diseño. Como te habrás dado cuenta, ahora tenemos un listado de estudiantes que buscar en otro listado, muy parecido al ejemplo anterior de los números comunes en dos listados, veamos el código:

```
let estudiantes = ["Juan","Maria"]
let cursoDiseno = ["Pedro","Maria","Diego","Juan","Paola"]
```

- ***Paso 5:*** Para validar si los estudiantes pertenecen al curso debemos comparar cada uno de estos en el curso y la forma de hacerlo es con ciclos anidados, veamos cómo quedaría el código.

```
let loEncontramos = false;
for (let j = 0; j <= cursoDiseno.length; j++) {
  for (let i = 0; i <= estudiantes.length; i++) {
      if(cursoDiseno[j] ===  estudiantes[i]) {
          console.log( estudiantes[i] + ' pertenece al curso de diseño')
      }
  }
```

```
}
```

Al ejecutar el código deberíamos ver lo siguiente:

```
Juan pertenece al curso de diseño
Maria pertenece al curso de diseño
```

Ejercicio propuesto (10)

Desarrollar un programa utilizando JavaScript que utilizando ciclos anidados calcule todas las tablas de multiplicar hasta 10.

Resumen

En esta lectura pudimos profundizar los conceptos que vimos anteriormente sobre ciclos y funciones, agregando elementos que nos otorgan herramientas para resolver problemas cada vez más complejos.

Conocimos aspectos relativos a la estructura de una función y sus tipos. Además, vimos que las variables tienen un alcance, lo que nos permite hacer que nuestro código sea seguro y no tenga comportamientos inesperados.

Revisamos los aspectos más relevantes de ES6, como las funciones flecha, que facilita la legibilidad del código y nos permite escribir las mismas instrucciones en menos líneas.

Finalmente, retomamos el concepto de ciclos y estructuras de control, pero esta vez los anidamos para resolver lógicas más avanzadas.

Como ves, hemos avanzado a pasos agigantados, utilizando recursos que manejamos y agregando detalles que nos han abierto un abanico de posibilidades. En programación por lo general ocurre esto: es necesario comprender las bases, porque sobre ellas se sustentan las lógicas que nos permitirán convertirnos en expertos programadores.

Solución de los ejercicios propuestos

1. Desarrollar un programa en JavaScript que calcule la resta de tres números enteros indicados en variables separadas previamente e indique el resultado directamente en la función de la resta, es decir, sin retorno de valor alguno.

```
var num1 = 1;
var num2 = 2;
var num3 = 3;

function restaNumeros (num1,num2,num3) {
    let resta = num1 - num2 - num3;
    alert("El resultado es: " + resta);
};

restaNumeros(num1,num2,num3);
```

2. Realizar un programa en JavaScript que calcule el volumen de un cilindro mediante una función. A dicha función se le deberán pasar los valores del radio y la altura. Luego debe mostrar el resultado del volumen dentro de la misma función sin retornar ningún tipo de valor. (Ayuda: volumen de un cilindro es: PI*radio^2*altura).

```
const PI = 3.1416;

function cilindro(radio,altura) {
     let volum = PI * radio * radio * altura;
     document.write(`El volumen del cilindro es: ${volum}`);
};

let radio = prompt("Ingrese el radio del cilindro");
let altura = prompt("Ingrese la altura del cilindro");
cilindro(radio,altura);
```

3. Desarrollar un programa en JavaScript donde se cree una variable global denominada “lenguaje” e inicie con el valor “JavaScript”, luego crear una función denominada “miFuncion” y dentro de ella mostrar primeramente con un console log el valor de la variable global, posteriormente crear otra variable dentro de la función denominada “lenguaje” e iniciar con el valor “NodeJS”, seguidamente mostrar en un console.log la nueva variable creada. Finalmente hacer el llamado a la función y luego mostrar el valor de la variable global con un console.log.

```
var languaje = "JavaScript";

function miFuncion () {
    console.log(languaje);
    var languaje =  "NodeJS";
    console.log(languaje);
};

miFuncion();
console.log(languaje);
```

4. Realizar un programa con JavaScript que calcula la resta de todos los términos disponibles en una variable del tipo arreglo con los siguientes datos numéricos: nums = [1, 2, 3, 5, 8, 13, 21].

```
const nums = [1, 2, 3, 5, 8, 13, 21];

function resta (nums) {
    if (nums.length < 1) {
        return 0;
    };
    var valor = nums.shift();
    return valor - resta(nums);
};
console.log(resta(nums));
```

5. Sin ejecutar el código, ¿cuál es el valor de `i` de los siguiente bloques de código ?

    5.1. *i = 10*
    5.2. *ReferenceError: i is not defined.*

6. Realizar un programa en JavaScript mediante el uso de funciones flecha que solicite al usuario ingresar su nombre y edad y retorne un saludo con el nombre, la edad e indicando si es mayor o menor de edad.

```
let saludar = () => {
      let nombre = prompt('Ingresa tu nombre');
      let edad = prompt('Ingresa tu edad');

      if (edad >= 18) {
      return `Hola ${nombre}, tu edad es: ${edad}, por lo tanto eres
mayor de edad`;
```

```
      }else{
      return `Hola ${nombre}, tu edad es: ${edad}, por lo tanto eres
menor de edad`;
      };
};

alert(saludar());
```

7. Mostrar los números impares y contar los números pares mediante un ciclo for para los números comprendidos entre el 1 y el 50 (incluidos ambos), pero cuando se llegue al número 11, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 40 el ciclo debe ser interrumpido y terminar su ejecución.

```
var impar = 0;

for (let i = 1; i <= 50; i++) {
    if (i == 11) {
      continue;
    };
    if (i == 40) {
      break;
    };
    if (i % 2 == 0) {
      document.write(i+"<br>");
    }else {
      impar++;
    };
};
document.write("La cantidad de números impares es: "+impar);
```

8. Realizar un programa con JavaScript mediante el ciclo while que muestre los números múltiplos de cinco y cuente los números que no sean múltiplo de cinco, para los números comprendidos entre el 0 y el 35 (incluidos ambos), pero cuando se llegue al número 10, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 30 el ciclo debe ser interrumpido y terminar su ejecución.

```
let no_multiplo = 0;
let i = 0;

while(i <= 35) {
    i++;
    if (i == 10) {
      continue;
    };
    if (i == 30) {
      break;
    };
    if (i % 5 == 0) {
      document.write(i+"<br>");
    }else {
      no_multiplo++;
    };
};
document.write("La cantidad de números no múltiplos de cinco es:
"+no_multiplo);
```

9. Realizar un programa con JavaScript mediante el ciclo do-while que muestre los números múltiplos de seis y cuente los números que no sean múltiplo de seis, para los números comprendidos entre el 2 y el 63 (incluidos ambos), pero cuando se llegue al número 19, el programa debe continuar y no mostrar el número en cuestión, igualmente, cuando se llegue al número 59 el ciclo debe ser interrumpido y terminar su ejecución.

```
let no_multiplo = 0;
let i = 2;

do {
    if (i == 19) {
      i++;
      continue;  // se ignora cualquier otro proceso
    };
    if (i == 59) {
      break;  // se rompe el ciclo actual
    };
    if (i % 6 == 0) {
      document.write(i+"<br>");
    }else {
      no_multiplo++;
```

```
    };
    i++;
}while(i <= 63);
document.write("La cantidad de números no múltiplos de seis es:
"+no_multiplo);
```

10. Utilizando ciclos anidados calcule todas las tablas de multiplicar hasta 10.

```
for (let i = 1; i <= 10; i++) {
      for (let j = 1; j <=10; j++) {
           console.log(`${j} x ${i}  = ${j*i}`);
      }
};
```